Санкт-Петербургский государственный университет

010300 Фундаментальные информатика и информационные технологии
Информационные технологии

Галактионов Вячеслав Аркадьевич

Организация сетевого взаимодействия между узлами в распределенной дисковой колоночной СУБД

Бакалаврская работа

Научный руководитель:
ассистент Чернышев Г. А.

Рецензент:
к. ф.-м. н., доцент СПбАУ Барашев Д. В.

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

010300 Fundamental Computer Science and Information Technology
Information Technologies

Viacheslav Galaktionov

Network interaction between nodes of a distributed column-store DBMS

Bachelor's Thesis

Scientific supervisor:
assistant professor George Chernishev

Reviewer:
associate professor at SPbAU Dmitry Barashev

Saint-Petersburg
2017

Оглавление

Введение

Интерактивная аналитическая обработка (OLAP) — один из популярных сценариев использования СУБД в индустрии [7, 13]. Данный сценарий используется для получения помощи в принятии бизнес-решений: генерации отчетов, анализа и предсказания продаж, оценки рисков и так далее.

Одна из основных особенностей такого сценария — редкое добавление новых данных в систему. В то же время необходимо иметь возможность исполнять сложные запросы над большими массивами данных за наименьшее время. Сам термин OLAP появился достаточно давно, в 1993 году [5], примерно в то же время было решено разделять два типа СУБД.

В начале XXI века область начала испытывать подъем, например в работе [4] указывается, что рынок OLAP систем в 1998 году оценивался всего в 2 миллиарда долларов США, а уже в 2006 году составлял 6 миллиардов долларов.

Одним из наиболее оптимальных способов организации СУБД, ориентированной на OLAP, является поколоночное хранение [2], позволяющее эффективно использовать аппаратные ресурсы системы [3].

Поддержка распределенности в том или ином виде в настоящее время является де-факто обязательным требованием к любой серьезной СУБД. Распределенность позволяет не только увеличивать объем хранимых данных, но и позволяет увеличивать эффективность выполнения запросов с помощью специальных алгоритмовс помощью специальных алгоритмов.

Аспект распределенности в классических (строковых) СУБД был подробнейшим образом изучен еще в 80-х. В то же время наблюдается дефицит или даже отсутствие исследовательских работ по распределенным колоночным СУБД. Конечно, к настоящему моменту было разработано несколько десятков индустриальных колоночных систем, однако научному сообществу про их внутреннее устройство практически ничего не известно. При этом, колоночные СУБД являются актуальной и востребованной в сообществе исследовательской темой. Дефицит публикаций объясняется просто: для проведения исследований в области распределенных колоночных СУБД необходимы специальные исследовательские прототипы таких систем. Функционально полных прототипов с открытым программным кодом на данный момент не существует [28].

При этом, использовать медиаторный подход [16] или расширять существующие централизованные СУБД нам не представляется оправданным в связи с имеющимися архитектурными ограничениями данных систем.

В связи с этим было принято решение с нуля начать разработку такой системы, которая получила название PosDB. Это касается и аспекта распределенности, которому и посвящена данная работа.

1. Постановка задачи

Целью данной работы является разработка и реализация архитектуры сетевого взаимодействия между узлами распределенной колоночной СУБД. Для достижения этой цели были выделены следующие подзадачи:

- Разработать протокол сетевого общения узлов.
- Спроектировать и реализовать схему сериализации пересылаемых по сети сущностей.
- Реализовать сетевое взаимодействие.
- Обеспечить восстановление оборванного соединения в контексте СУБД.
- Исследовать производительность полученной архитектуры.

2. Существующие подходы

Существует множество различных моделей организации сетевого взаимодействия между узлами распределенной системы [16, 19]:

- Одноранговые,
- Клиент-серверные,
- Многоуровневые.

Узлы одноранговой сети программно идентичны друг другу, то есть имеют одинаковые права и функции. Конечный пользователь может подключиться к любому из узлов и передать ему свой запрос на обработку. Данные могут быть распределены по такой сети произвольным образом.

В клиент-серверных системах есть два типа узлов: клиентские и серверные. Основной задачей первых является обработка пользовательских запросов, а вторых — управление данными. В таких системах есть несколько вариантов исполнения запроса: поставкой запроса, данных и гибридный. Идея первого подхода заключается в передаче клиентом плана запроса серверу для исполнения. Когда запрос будет выполнен, клиент заберет результат и отправит его пользователю. В случае второго подхода серверы используются только как хранилища данных: во время исполнения запроса клиент будет запрашивать притягивать себе нужные данные. Гибридный подход является более общим и объединяет описанные выше. Более подробное описание и сравнение приведено в работе [16].

В многоуровневой системе узлы составляют иерархию: “родительские” узлы выступают в роли серверов для своих “потомков”, которые, в свою очередь, обслуживают своих. Такой подход часто применяется для создания так называемых трехуровневых систем, где функционал распределяется между тремя типами машин: первые отвечают за взаимодействие с пользователем, вторые — за обработку бизнес-логики, а третьи — за управление данными.

Также многоуровневые архитектуры реализуются в системах, объединяющих различные СУБД в одну распределенную систему. Для достижения этой цели разрабатывается специальное связующее программное обеспечение (middleware), задача которого состоит в координации работы отдельных СУБД, которые могут иметь абсолютно разные свойства и строение. Например, частями такое системы могут быть как обыкновенные реляционные базы, так и базы, работающие по принципу “ключ-значение”.

Помимо этого существует и другая классификация распределенных систем, основанная на разделении ресурсов [19, 24]:

- Shared-Everything (SE): процессоры разделяют между собой оперативную память и диск;

- Shared-Disk (SD): каждый процессор имеет доступ только к своей части оперативной памяти, но сохраняют доступ ко всему диску;
- Shared-Nothing (SN): каждый процессор собственную оперативную память и диск;
- Clustered-Everything (CE): SE-кластеры соединены по принципу SN;
- Clustered-Disk (CD): SD-кластеры соединены по принципу SN.

Во время проектирования PosDB было решено использовать подход с одноранговой сетью, так как он является наиболее общим [16], что позволит реализовать и исследовать различные сценарии работы. Также в процессе построения системы считается, что узлы не разделяют никаких ресурсов, так как такая архитектура — наиболее предпочтительна для построения масштабируемых высокопроизводительных систем [14, 26].

Существует множество способов реализации такого взаимодействия. Например, в работе [6] описываются и сравниваются следующие:

- Распределенная общая память (DSM),
- Использование технологии, основанной на RPC,
- Использование одной из реализаций MPI,
- Ручное сокетное программирование.

Существуют и другие работы [10, 21, 22], приводящие обзор этих подходов и их альтернатив. Однако в приведенной выше классификации отражены все наиболее общие варианты.

Тем не менее, в рамках этой дипломной работы была поставлена задача реализовать сетевую подсистему путем сокетного программирования. Помимо обучающего аспекта этого подхода, такая реализация позволяет сократить число внешних зависимостей проекта и добиться большей гибкости работы. Также это позволит добиться большей производительности, избежав накладных расходов, связанных с существующими решениями.

3. Выполнение запроса в централизованном случае

Для лучшего понимания проделанной работы имеет смысл кратко рассмотреть модель выполнения запросов в централизованном случае. В этом разделе будет приведено краткое описание работы PosDB, более детальное описание можно найти в работах [20, 1, 28, 29].

PosDB реализует итераторную модель Volcano [11, 12]. В ней физический план запроса представляется в виде дерева физических операторов, каждый из которых реализует интерфейс итератора. Каждый оператор производит какую-то манипуляцию над данными, полученными от его потомков. Для улучшения производительности данные передаются не по одной записи, а массивами.

Для описания структуры базы данных используется каталог, в котором содержатся списки:

- отношений,
- атрибутов каждого отношения,
- горизонтальных фрагментов каждой колонки,
- реплик (replication) каждого горизонтального фрагмента.

Для каждого горизонтального фрагмента указаны соответствующие диапазоны позиций, а для каждой реплики — адрес хранящего узла (в случае локального хранения будет указан текущий узел — localhost).

Важной идеей в колоночных системах является материализация, или процесс сборки кортежей из набора колонок. Существует два основных типа материализации [17]:

- Ранняя: сборка кортежей происходит как можно раньше, при первой необходимости использовать соответствующие данные;
- Поздняя: кортежи собираются как можно позже, иногда даже уже во время возвращения результата пользователю.

Так как ранняя материализация уже хорошо изучена [29], в PosDB было решено реализовать позднюю. Более конкретно, данные материализуются после последней операции соединения.

3.1. Пример

Рассмотрим один из запросов использованного в экспериментах эталонного теста производительности SSB [18], в качестве примера. SQL-код этого запроса представлен на листинге 1.

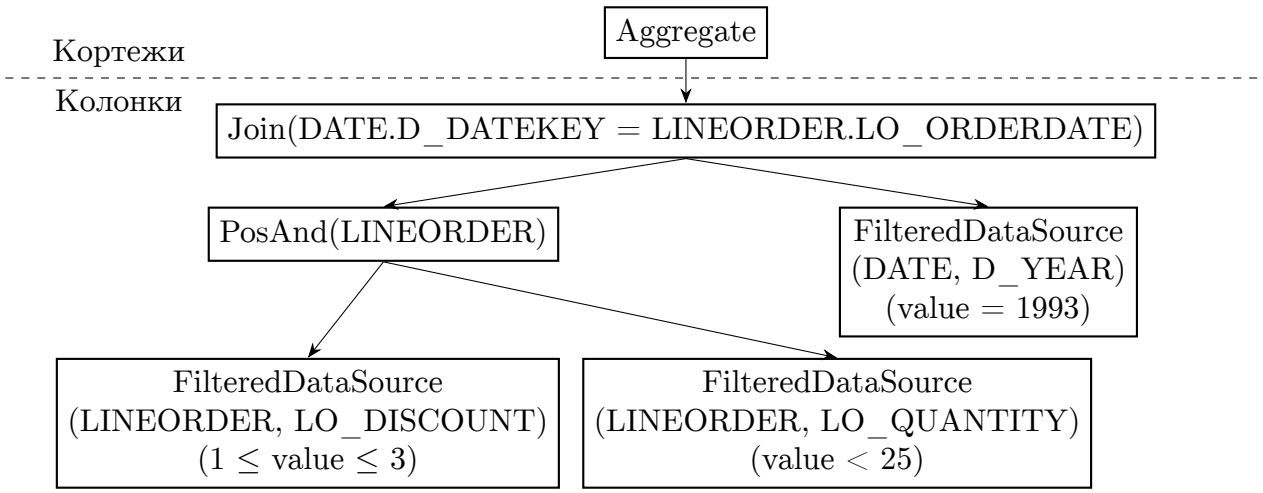


Рис. 1: Пример плана запроса (в упрощенном виде)

На рис. 1 изображен итоговый план этого запроса. Пунктирная линия — момент материализации. В данном случае работа над материализованными данными производится одним оператором — `Aggregate`, который в качестве одного из параметров принимает необходимую программу вычислений. До материализации используется более сложное дерево, состоящее из одной операции соединения.

Фильтрация колонки осуществляется с помощью оператора `FilteredDataSource`, принимающего идентификатор колонки и предикат. Для фильтрации отношения по нескольким атрибутам используется оператор `PosAnd`, основная задача которого — пересечение множеств позиций, получаемых от операторов-потомков. Здесь он используется для получение позиций, которым соответствуют кортежи, удовлетворяющие обоим предикатам.

3.2. До материализации

Запрос здесь представляется в виде дерева операторов, каждый из которых предоставляет интерфейс, описанный на листинге 2:

`PosBinding` — структура, соотносящая идентификатор таблицы с набором позиций в ней. Получив очередной экземпляр такой структуры, оператор извлекает нужные ему массивы позиций и обрабатывает их, в итоге возвращая измененный `PosBinding`.

```
1 select sum(lo_extendedprice*lo_discount) as revenue
2 from lineorder, date
3 where lo_orderdate = d_datekey
4 and d_year = 1993
5 and lo_discount between 1 and 3
6 and lo_quantity < 25;
```

Листинг 1: Код запроса

```
1  class Operator {
2  public:
3      virtual PosBinding *getNext() = 0;
4      virtual ~Operator(){}
5      virtual void rewind() = 0;
6  };
```

Листинг 2: Интерфейс оператора, используемого до материализации

Возвращение нулевого указателя сигнализирует о конце потока данных.

Очевидно, не все операции можно провести, зная только позиции. Для получения соответствующих данных в PosDB вводится понятие считывателя (reader). Эта структура инициализируется идентификатором читаемой колонки, после чего принимает массив позиций, по которому возвращает массив данных.

3.3. После материализации

На этом этапе работа проводится уже над кортежами атрибутов, а не над колонками. Иначе говоря, над данными из разных колонок, соответствующими одной строке экземпляра PosBinding, ставшего результатом первого этапа.

Вместо операторов, описанных в предыдущем подразделе, эта часть плана запроса исполняется операторами, реализующими интерфейс, представленный на листинге 3.

Основное отличие — в передаваемых между операторами объектах. Если до материализации передаются PosBinding, то здесь — кортежи, представленные указателем на байтовый массив. Доступ к элементам предоставляется с помощью объектов TupleHeader, хранящих типы элементов и их смещения внутри массива.

Помимо основных функций, отвечающих за возвращение результата, также предоставляются операции вывода результата и подсчета различных хеш-функций.

```
1  class TupleOperator {
2  public:
3      virtual TupleHeader *getHeader() const = 0;
4      virtual byte *getNext() = 0;
5      virtual std::vector<byte *> getTuples();
6      void print(std::ostream &ost);
7      void process(std::ostream &ost);
8      size_t getSumHash();
9      size_t getAdlerHash();
10     virtual void rewind() = 0;
11     virtual ~TupleOperator(){}
12 };
```

Листинг 3: Интерфейс оператора, используемого после материализации

4. Реализация

Для решения поставленных задач был выбран язык программирования C++. Такой выбор вполне естественен, учитывая, что вся кодовая база PosDB была написана на этом языке.

В качестве сетевого протокола был выбран TCP, так как для распределенной СУБД важны сохранность передаваемых пакетов и их порядок. Разумеется, возможности протокола далеко не безграничны, и некоторые возможности, связанные, в основном, с обработкой сетевых ошибок, необходимо реализовать в рамках самой системы.

Для обеспечения распределенного исполнения запроса на нескольких узлах были реализованы следующие сущности:

1. Сетевой считыватель `NetworkReader`,
2. Сетевой оператор `ReceivePos`,
3. Сервер, управляющий входящими подключениями.

Помимо этого была разработана система сериализации для передачи различных объектов: планов запроса, экземпляров PosBinding, массивов позиций и значений.

4.1. Основные понятия

Для начала рассмотрим предполагаемую схему работы системы при обработке запроса на нескольких узлах.

1. Пользователь подключается к одному из узлов и передает свой запрос.
2. Этот узел проводит первоначальную обработку, получая в итоге план запроса.
3. Узел начнет исполнять получившийся план.
4. Как только начнет использоваться `NetworkReader` или `ReceivePos`, этот узел, далее называемый клиентом, подключится к нужному, далее называемому сервером.
5. Клиент и сервер взаимодействуют по описанным далее алгоритмам.
6. Остальная часть плана выполняется как в централизованном случае.

Выбор узла производится в зависимости от используемой сущности: `NetworkReader` использует информацию, хранящуюся в каталоге, а `ReceivePos` получает адрес узла в качестве параметра.

4.2. Принцип работы сервера

Основным методом реализации сетевых операторов и считывателей является “оборачивание”. На стороне клиента используются специальные реализации этих сущностей, функции-члены которых посылают запрос серверу вместо самостоятельного выполнения соответствующей работы.

В качестве рабочих единиц, обслуживающих клиентские запросы были выбраны потоки. Другим часто используемым подходом является применение процессов, но их сложнее координировать, если необходимо восстанавливать соединения.

Основной поток сервера принимает новые соединения и создает для каждого по потоку. После этого клиент сообщает, какую сущность хочет использовать, и поток вызывает статическую функцию `handle` соответствующего класса (конкретные примеры будут описаны далее), которая займется обработкой нового подключения.

Далее новый поток начинает ожидать от клиента команд, представленных числовыми идентификаторами.

4.3. Сериализация

Для пересылки данных лучше всего упаковать их в один непрерывный буфер. Именно в этом и состоит задача сериализации.

Альтернативой было бы отправлять данные по частям. Например, вместо упаковки дерева запроса в буфер можно отправлять операторы по очереди во время обхода дерева. Проблема такого подхода заключается в многократных системных вызовах пересылки данных, что повлечет за собой значительные накладные расходы.

Чтобы объекты, которые необходимо пересылать по сети, можно было единообразно сериализовать, они должны реализовывать интерфейс, указанный на листинге 4.

Как видно из определения, функции-члена, отвечающей за десериализацию, нет. Вместо этого, воссоздание объекта производится в специальной статической функции-члене, определяемой в каждом классе, реализующем этот интерфейс.

Основной причиной такой реализации является удобство использования. Включение такой функции в интерфейс означало бы возможность существования объекта в недействительном состоянии, что требует значительных объемов кода для поддержки и обработки.

Еще одной альтернативной реализацией мог бы стать конструктор, принимающий

```
1 struct Serializable {
2     virtual size_t sizeInBytes() const = 0;
3     virtual byte *serialize  (byte *buf) const = 0;
4 };
```

Листинг 4: Интерфейс сериализуемых объектов

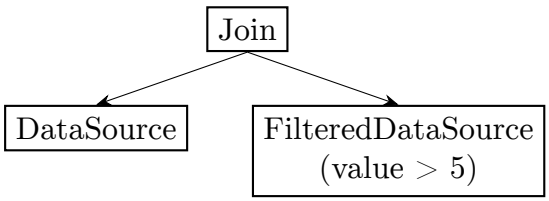


Рис. 2: Пример сериализуемого запроса

указатель на сериализованные данные. Проблема такого подхода — невозможность десериализации абстрактных классов, которая является частым сценарием использования.

Рассмотрим пример сериализации и десериализации для конкретного плана запроса с рис. 2. Этот простой запрос состоит из соединения двух источников данных, к одному из которых привязан предикат, используемый для фильтрации данных.

Каждому классу операторов присвоен свой числовой идентификатор. Например, у источников данных это “0”, а у операторов соединения — “1”. При сериализации оператор прежде всего указывает свой класс с помощью такого идентификатора, после чего идут дополнительные данные: идентификаторы используемых колонок, сериализованные операторы-потомки и так далее.

Предикаты, использующиеся в `FilteredDataSource`, тоже представляются в виде дерева, так что их сериализация производится по аналогичной схеме.

Результатом этого процесса будет буфер следующего вида: идентификатор операторов соединения, описание используемых колонок, идентификатор источника данных, описание колонки, сигнал отсутствия предиката, идентификатор источника данных, описание колонки, сериализованный предикат.

При десериализации этого плана будет вызвана функция `Operator::deserialize`, которая извлечет идентификатор типа оператора и вызовет функцию-член соответствующего класса.

4.4. Управление подключением и обнаружение ошибок

Для более удобной работы с сетью был разработан специальный класс `Connection`, инкапсулирующий сетевое соединение. В его функционал входят установление соединения, пересылка данных и обнаружение ошибок сети.

Основные функции пересылки данных “оборачивают” системные вызовы `send` и `recv` и принимают указатель на массив данных и его длину. Также были добавлены вспомогательные перегрузки, принимающие различные сущности: значения примитивных типов, строки и так далее.

Наиболее важной и сложной возможностью этого класса является обнаружение разрывов соединения. Сам протокол TCP в состоянии справиться с этой задачей, но это займет много времени, если использовать настройки по умолчанию.

В связи с этим было решено ограничить время ожидания получения и отправки данных с помощью двух параметров сетевого сокета: `SO_RCVTIMEO` и `SO_SNDTIMEO`.

В качестве максимального времени ожидания было решено взять промежуток в 30 секунд. Такой выбор обосновывается тем, что этого достаточно для почти любой операции, и в то же время позволит системе достаточно быстро адаптироваться к изменениям в сети.

Разумеется, предусматриваются случаи, когда операция длится дольше этого лимита. Подробнее это будет описано в разделе 4.6.

Чтобы избежать разрыва соединения во время простоя оператора (пока выполняется другая часть плана), на стороне клиента создаются отдельные потоки, задача которых — отправлять серверу специальные сообщения. Эти сообщения содержат в себе код "пустой" команды, которая даст серверу понять, что клиент ещё в сети.

4.5. Сетевой считыватель: NetworkReader

Как было сказано в разделе 2, операторы PosDB используют считыватели для получения данных на позициях, полученных от дочернего оператора. В случаях, когда данные лежат на одном узле сети, но должны обрабатываться на другом, требуются считыватели, которые бы отправляли массивы позиций и получали массивы данных.

Именно такое поведение и реализовано в классе `NetworkReader`, реализующем интерфейс обычного считывателя. Рассмотрим его сетевые функции:

- `Update(PosBinding*)` служит для обновления текущего массива позиций. Так как PosBinding описывает несколько колонок, достаточно переслать только один массив позиций, значительно сокращая объем передаваемых данных;
- `getNextVB()` запрашивает новую часть данных и принимает ее.

Легко заметить простоту работы и отсутствие сопутствующих сложных вычислений. Состояние считывателя состоит из идентификатора колонки, одного массива позиций и номера текущей части данных. Именно благодаря этим свойствам сетевой считыватель не нуждается в сложной обработке разрыва соединения: достаточно восстановить соединение и переслать это состояние.

Для сокращения объема пересылаемых данных была реализована команда, отсутствующая в интерфейсе обычных считывателей: пропуск указанного объема данных. В качестве параметра сервер получает количество вызовов `getNextVB`, которые необходимо совершить.

4.6. Сетевой оператор: ReceivePos

4.6.1. Принцип работы

Для исполнения произвольного дерева операторов на удаленном узле необходимо использовать специальный оператор ReceivePos. В качестве параметров он получает адрес нужного сервера и поддерево, которое необходимо там исполнить.

Для своего предка в дереве операторов `ReceivePos` выглядит так, словно никакой сети нет, и работа ведется напрямую с деревом. Это достигается с помощью подхода "оборачивания", описанного ранее: `ReceivePos` реализует интерфейс обычного оператора.

Сложно оценить, сколько времени потребуется для получения следующего результата. Некоторые операции могут занять больше времени, чем позволяет временное ограничение на получение новых данных. В таком случае принимающая сторона решит, что появились проблемы с сетью, и переподключится.

Такие ложные срабатывания могут только мешать, если использовать более сложные стратегии восстановления соединения, которые включали бы в себя выбор другого узла. После подключения к другому узлу придётся восстанавливать утерянное состояние всех операторов поддерева. Чтобы их избежать, во время исполнения поддерева, сервер с определенным интервалом отправляет клиенту сообщения, что работа еще исполняется. Как только результат будет готов к отправке, клиент получит соответствующее сообщение.

4.6.2. Восстановление после ошибок сети

Учитывая, что дерево-потомок может быть произвольно сложным, можно заключить, что подход к восстановлению сетевого соединения, описанный в разделе 4.4, здесь не подходит. Восстановление нужного состояния для заново пересланного дерева может занимать долгое время, а в условиях ненадежной сети, где обрывы соединения случаются достаточно часто, процесс может застрять в бесконечном цикле пересылки и пропуска нужного объема результатов.

В связи с этим было решено хранить план на стороне сервера и позволить клиенту использовать его в рамках нескольких последовательных сетевых соединений. Это было реализовано с помощью механизма сессий [27].

Когда клиент подключается к серверу и передает свое поддерево, сервер генерирует для него уникальный идентификатор, который клиент сможет использовать для возвращения к переданному ранее плану.

Вместе с этим клиент при переподключении передает номер последнего полученного результата. Возможны три ситуации:

1. Сервер получает 0, что сигнализирует о сетевой ошибке во время вызова `rewind`.

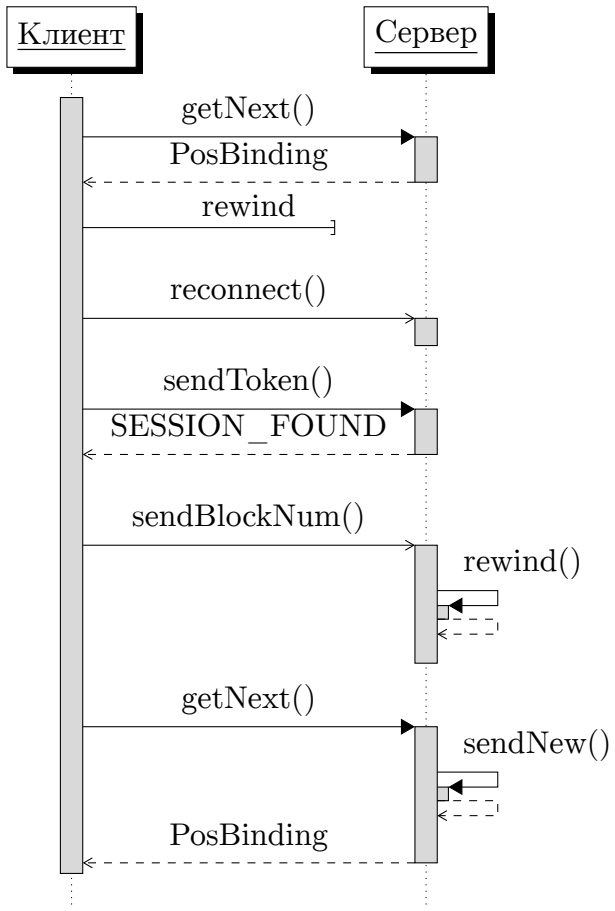


Рис. 3: Восстановление провального `rewind`

В таком случае на стороне сервера вызывается соответствующая функция.

2. Клиентское значение на единицу меньше серверного. Это является признаком того, что предыдущий результат не был доставлен, и будет переслан заново.

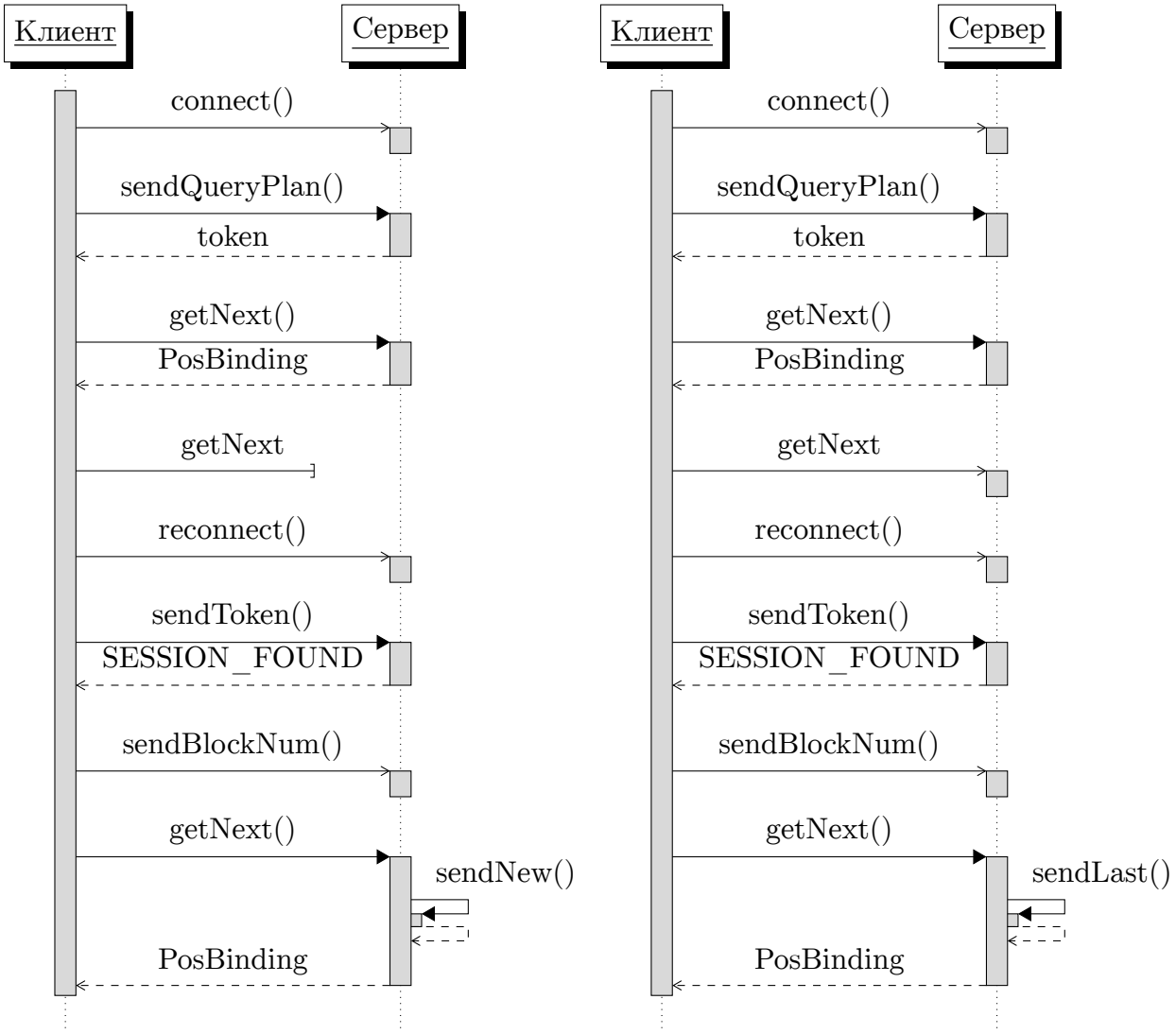
3. Клиентское и серверное значение совпадают. Это говорит о том, что обе стороны синхронизированы и что дополнительных действий не требуется.

Схема этих трех ситуаций изображены на рис. 3 и 4.

4.6.3. Восстановление после ошибок сервера

Если используемый сервер пропал из сети (например, после неожиданной перезагрузки или снятия соответствующего процесса), то текущая сессия будет потеряна целиком. Клиент переподключится к старому серверу или подключится к новому и отправит идентификатор своей сессии, на что получит ответ, сигнализирующий об отсутствии такой сессии.

Далее сервер создаст новую сессию с этим идентификатором. Клиенту придется переслать свое поддерево и порядковый номер результата, который он ожидает полу-



(a) Потеря запроса

(b) Потеря ответа

Рис. 4: Обработка сетевых ошибок

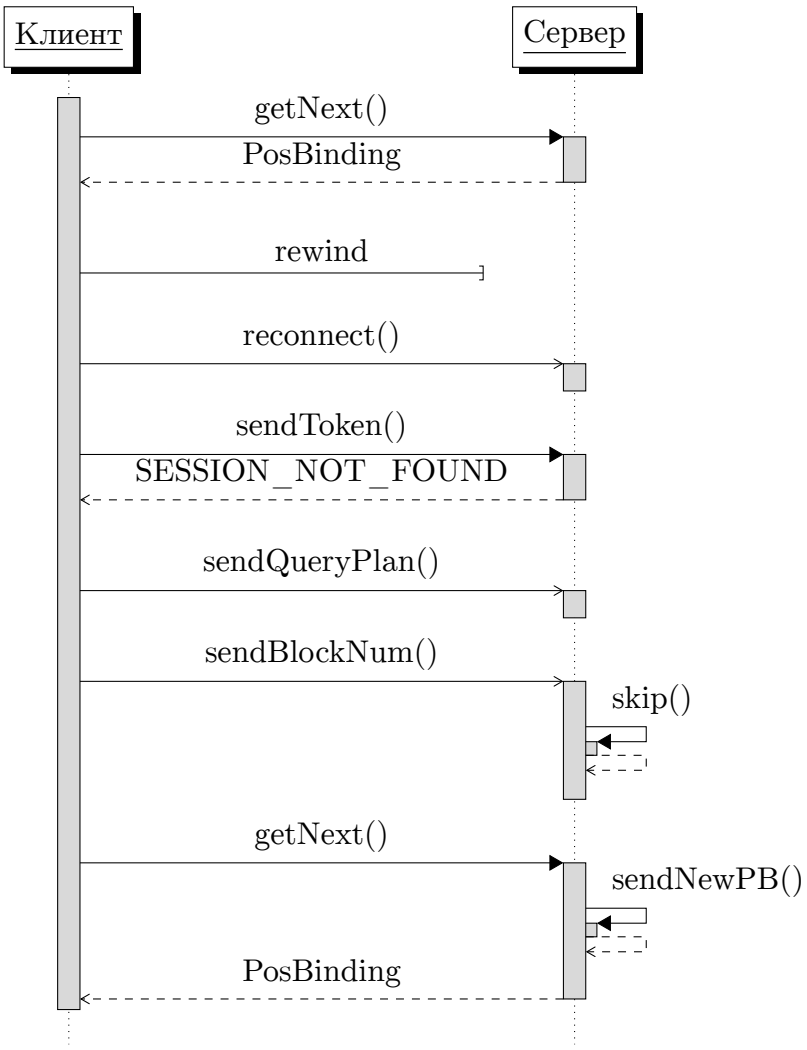


Рис. 5: Восстановление после ошибок сервера

чить. Сервер совершит необходимое число вызовов `getNext()` для полученного плана и будет готов обрабатывать дальнейшие команды.

Схема такого процесса изображена на рис. 5.

5. Эксперименты

Для оценки работоспособности и производительности полученной системы были проведены замеры времени работы с использованием эталонного теста производительности Star Schema Benchmark [18].

Этот тест основан на TPC-H [25] и имеет следующую структуру:

- 5 таблиц: CUSTOMER, DATE, LINEORDER, PART, SUPPLIER;
- 4 запроса;
- 3–4 варианта каждого запроса с разным объемом результата.

Тестовая база данных построена на основе схемы звезды, то есть имеет одну таблицу фактов и несколько таблиц измерений. Размер первой значительно превосходит размер остальных.

В связи с этим было решено распределить данные следующим образом (см. рис. 6): таблицы измерений реплицированы на всех узлах сети, а таблица фактов разбита на фрагменты одинакового размера, каждый из которых перемещен на свой узел.

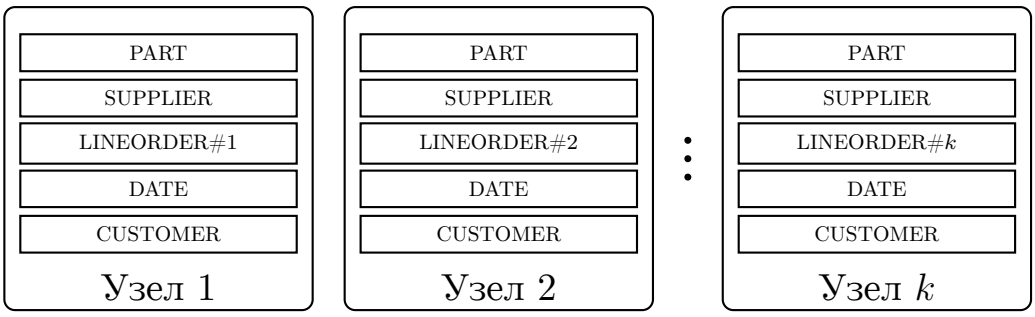


Рис. 6: Схема распределения данных

Для каждого запроса был создан распределенный вариант. Для каждого узла сети создается соответствующий ему оператор `ReceivePos`, получающий свое поддерево, задача которого — обработать часть данных, расположенных на узле. Эти поддеревья объединяются с помощью оператора `UnionAll`. Пример такого распределения для запроса с рис. 1 изображен на рис. 7.

Для экспериментов использовались восемь машин следующей конфигурации:

- Intel(R) Core(TM) i5-2310 CPU @ 2.90GHz (4 физических ядра),
- 4 ГБ ОЗУ,
- Ubuntu Linux 16.04.1 (64 разряда),
- GCC 5.4.0.

В работе [1] подробно описываются эксперименты, связанные с метриками ускорения и масштабируемости.

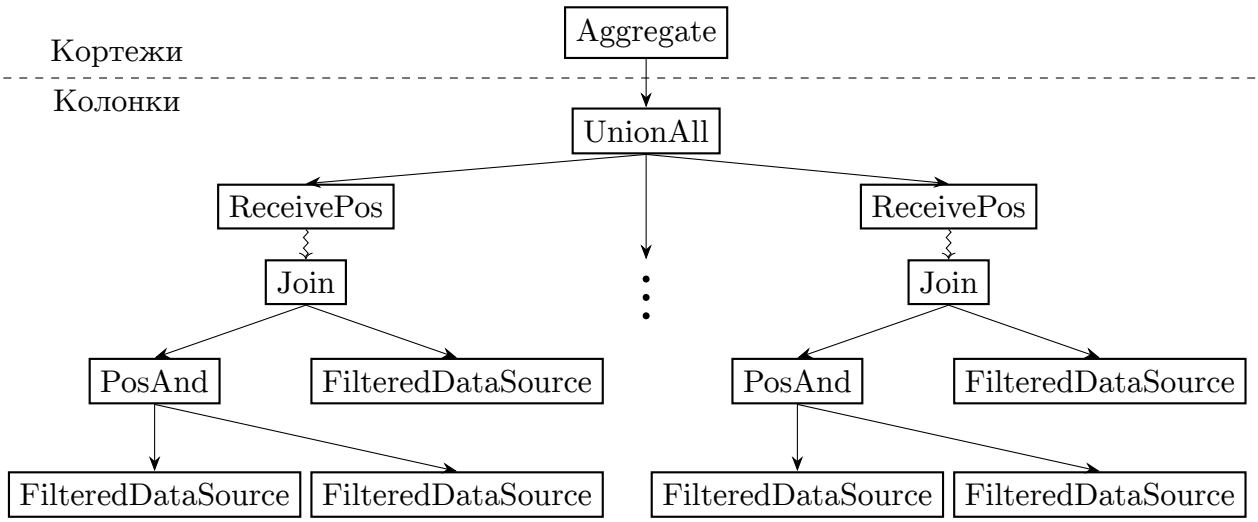


Рис. 7: Схема распределения запроса

5.1. Ускорение

Одной из наиболее интересных метрик производительности распределенной системы является ускорение (speed up) [8, 9, 23, 15]. Его можно рассчитать по следующей формуле:

$$\text{Ускорение} = \frac{\text{производительность k узлов на x данных}}{\text{производительность 1 узла на x данных}}$$

В качестве нагрузки для этого эксперимента были использованы данные эталонного теста производительности SSB со значением фактора масштаба (scale factor) равным 50. Это соответствует примерно 10 гигабайтам в формате системы PosDB.

При разработке параллельных систем стремятся добиться линейного ускорения [15]. В крайне редких случаях удается добиться сверхлинейного (superlinear) ускорения, но чаще всего оно является следствием особенностей архитектуры или использования

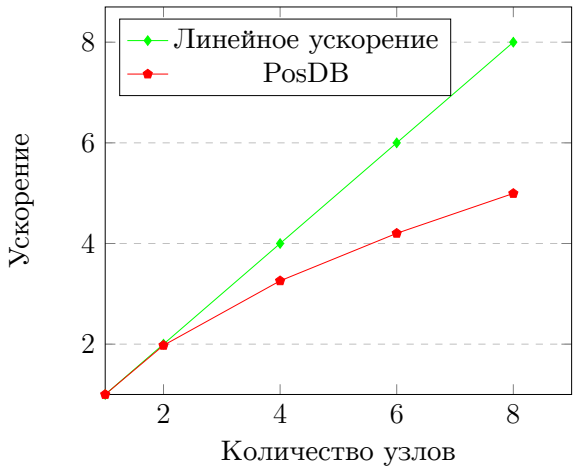


Рис. 8: График ускорения при увеличении количества узлов

неоптимальных последовательных алгоритмов.

Результаты замеров представлены на рис. 8. Как можно заметить, производительность системы увеличивается с добавлением узлов. Очевидно, полученное ускорение нелинейно. Это объясняется тем, что система находится на раннем этапе своего развития и еще не обладает важными для производительности возможностями.

5.2. Масштабируемость

Вместе с ускорением была исследована метрика масштабируемости (scale up) [8, 15]. Для ее подсчета используется следующая формула:

$$\text{Scaleup} = \frac{\text{производительность k узлов на x * k даных}}{\text{производительность 1 узла на x данных}}$$

Эксперимент проводился с использованием данных, сгенерированных с разными факторами масштаба: 10, 20, 40, 60, 80. Каждый набор данных обрабатывался соответствующим количеством машин: 1, 2, 4, 6, 8.

Результаты замеров представлены на рис. 9. Так же как и в предыдущем разделе, на графике изображен и лучший случай: линейная масштабируемость. Как и в случае с ускорением, сверхлинейная масштабируемость возможна, но крайне редка.

Помимо лучшего, на графике отмечен и худший. Под худшим случаем здесь понимается ситуация, когда добавление новых узлов никак не влияет на производительность системы.

Разумеется, PosDB не достигает этого идеала, но имеет большую масштабируемость, чем в худшем случае.

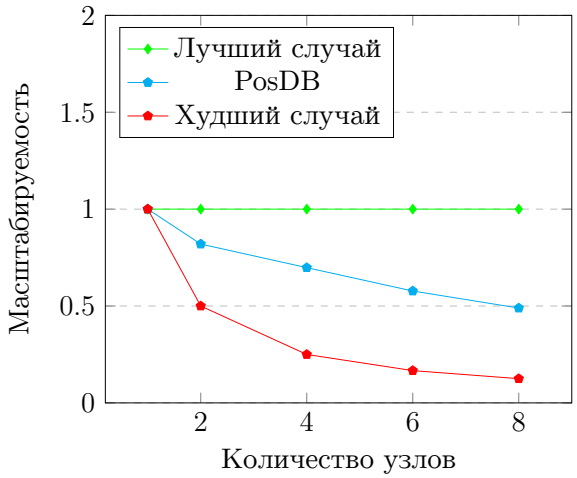


Рис. 9: График масштабируемости при увеличении количества узлов

5.3. Накладные расходы

Для измерения затрат, связанных с поддержкой сети, были проведены замеры времени работы в следующих ситуациях:

1. Централизованное исполнение;
2. Сервер запущен на той же машине, что и клиент;
3. Сервер и клиент запущены на двух разных машинах.

В качестве нагрузки были использованы данные эталонного теста производительности SSB со следующими факторами масштаба: 1, 2, 4, 6, 8, 10. Важно отметить, что во всех случаях данные не фрагментировались, то есть каждый запрос обрабатывался одним узлом без использования параллелизма. Результаты замеров изображены на рис. 10.

Полностью централизованный случай занял наименьшее время. Это вполне ожидаемый результат, объясняемый отсутствием каких-либо накладных расходов, связанных с сетью.

Примечательно, что выполнение обеих ролей на одной машине занимает больше времени, чем на двух. Оправданно было бы ожидать обратного, ведь к затратам на подготовку сообщений добавляется пересылка данных по сети. Однако благодаря высокой пропускной способности сети эта дополнительная нагрузка становится незначительной, а при исполнении на двух узлах подготовка и обработка сообщений распределяется между двумя машинами, таким образом уменьшая общее время выполнения.

В целом можно сказать, что затраты на сеть достаточно невелики, хоть и заметны.

5.4. Восстановление соединений

Для оценки работоспособности восстановления соединений была реализована эмуляция нестабильной сети. Для достижения такого эффекта без временных задержек, требуемых для определения проблем с сетью, в функции класса `Connection`, связанные с передачей данных, были добавлены разрыв соединения и возвращение ошибки. Очевидно, такое поведение не должно проявляться при пересылке каждого сообщения. В связи с этим был использован генератор псевдослучайных чисел, чтобы обеспечить ошибку при пересылке в среднем одного сообщения из десяти.

В качестве нагрузки для системы в этом эксперименте был так же использован эталонный тест производительности SSB с единичным значением фактора масштаба.

Отдельно были исследованы два случая: исполнение запроса на сервере с передачей только позиций и исполнение запроса на клиенте с передачей только данных. В течение эксперимента системе пришлось восстанавливаться несколько раз в секунду.

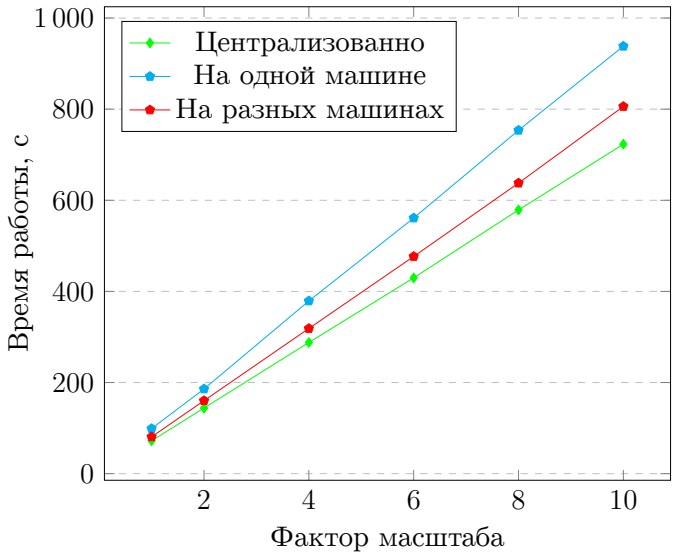


Рис. 10: Время работы в разных сетевых конфигурациях (в секундах)

Несмотря на это, запросы были выполнены успешно, были получены те же результаты, что и в локальном случае. Ожидаемо, время работы значительно возросло.

Заключение

В рамках данной работы были достигнуты следующие результаты:

- В рамках сетевой подсистемы реализованы:
  - сериализация различных сущностей: массивов данных, планов запроса, различных идентификаторов, выражений и предикатов,
  - фрагментирование запроса и рассылка его по узлам-исполнителям,
  - считывание данных с удаленных узлов,
  - отслеживание сетевых ошибок и восстановление соединений.
- Экспериментально исследована построенная РСУБД с использованием эталонного теста производительности SSB:
  - ускорение,
  - масштабируемость,
  - сетевые накладные расходы,
  - эффективность реализованного подхода к восстановлению соединений.

Список литературы

[1] A study of PosDB Performance in a Distributed Environment / George Chernishev, Viacheslav Galaktionov, Valentin Grigorev et al. // Proceedings of the 2017 Software Engineering and Information Management. — SEIM '17. — Saint-Petersburg, Russia, 2017. — 6 p.

[2] Abadi Daniel J., Boncz Peter A., Harizopoulos Stavros. Column-oriented Database Systems // Proc. VLDB Endow. — 2009. — Aug. — Vol. 2, no. 2. — P. 1664–1665. — Access mode: https://doi.org/10.14778/1687553.1687625.

[3] Abadi Daniel J., Madden Samuel R., Hachem Nabil. Column-stores vs. Row-stores: How Different Are They Really? // Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. — SIGMOD '08. — New York, NY, USA : ACM, 2008. — P. 967–980. — Access mode: http://doi.acm.org/10.1145/1376616.1376712.

[4] Chernishev George. Physical design approaches for column-stores // SPIIRAS Proceedings. — 2013. — Vol. 30. — P. 204–222. — online; accessed: www.mathnet.ru/trspy682.

[5] Codd E.F., Codd S.B., Salley C.T. Providing OLAP (On-line Analytical Processing) to User-analysts: An IT Mandate. — Codd & Associates, 1993. — Access mode: https://books.google.ru/books?id=pt0lGwAACAAJ.

[6] Costea Andrei, Ionescu Adrian. Query Optimization and Execution in Vectorwise MPP : Master's thesis / Andrei Costea, Adrian Ionescu ; Vrije Universiteit Amsterdam. — 2012. — Aug.

[7] Date C.J. An Introduction to Database Systems. — 8 edition. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003. — ISBN: 0321197844.

[8] DeWitt David, Gray Jim. Parallel Database Systems: The Future of High Performance Database Systems // Commun. ACM. — 1992. — Jun. — Vol. 35, no. 6. — P. 85–98. — Access mode: http://doi.acm.org/10.1145/129888.129894.

[9] Eager D. L., Zahorjan J., Lazowska E. D. Speedup versus efficiency in parallel systems // IEEE Transactions on Computers. — 1989. — Mar. — Vol. 38, no. 3. — P. 408–423.

[10] Golub Marin, Jakobović Domagoj, Janeš Ivan. An overview of distributed programming techniques // Hypermedia and Grid Systems, MIPRO 2005. — 2005.

[11] Graefe Goetz. Query Evaluation Techniques for Large Databases // ACM Computing Surveys. — 1993. — June. — Vol. 25, no. 2. — P. 73–169.

[12] Graefe G. Volcano — An Extensible and Parallel Query Evaluation System // IEEE Trans. on Knowl. and Data Eng. — 1994. — Feb. — Vol. 6, no. 1. — P. 120–135. — Access mode: http://dx.doi.org/10.1109/69.273032.

[13] Hector Garcia-Molina, Jeffrey D. Ullman, Jenifer Widom. Database Systems. The Complete Book. — 2nd edition. — Pearson Education, Inc, 2008.

[14] Hellerstein Joseph M., Stonebraker Michael, Hamilton James. Architecture of a Database System // Found. Trends databases. — 2007. — Feb. — Vol. 1, no. 2. — P. 141–259. — Access mode: http://dx.doi.org/10.1561/1900000002.

[15] High Performance Parallel Database Processing and Grid Databases / David Taniar, Clement H. C. Leung, Wenny Rahayu, Sushant Goel. — Wiley Publishing, 2008. — ISBN: 0470107626, 9780470107621.

[16] Kossmann Donald. The State of the Art in Distributed Query Processing // ACM Comput. Surv. — 2000. — Dec. — Vol. 32, no. 4. — P. 422–469. — Access mode: http://doi.acm.org/10.1145/371578.371598.

[17] Materialization Strategies in a Column-Oriented DBMS / D. J. Abadi, D. S. Myers, D. J. DeWitt, S. R. Madden // 2007 IEEE 23rd International Conference on Data Engineering. — 2007. — April. — P. 466–475.

[18] O'Neil P. E., O'Neil E. J., Chen X. The Star Schema Benchmark (SSB). — 2007.

[19] Ozsu M. Tamer. Principles of Distributed Database Systems. — 3rd edition. — Upper Saddle River, NJ, USA : Prentice Hall Press, 2007. — ISBN: 9780130412126.

[20] PosDB: a Distributed Column-Store Engine / George Chernishev, Viacheslav Galaktionov, Valentin Grigorev et al. // Proceedings of the 2017 A.P. Ershov Informatics Conference. — PSI '17. — Saint-Petersburg, Russia, Forthcoming. — 7 p.

[21] Qureshi Kalim, Rashid Haroon. A performance evaluation of rpc, java rmi, mpi and pvm // Malaysian Journal of Computer Science. — 2005. — Vol. 18, no. 2. — P. 38–44.

[22] SALIH SHADMAN. Selection of Computer Programming Languages for Developing Distributed Systems : Ph. D. thesis / SHADMAN SALIH ; De Montfort University. — 2014.

[23] Parallel Computing: Performance Metrics and Models : Rep. ; Executor: Sartaj Sahni, Venkat Thanvantri : 1995.

[24] Sokolinsky L. B. Survey of Architectures of Parallel Database Systems // Program. Comput. Softw. — 2004. — Nov. — Vol. 30, no. 6. — P. 337–346. — Access mode: http://dx.doi.org/10.1023/B:PACS.0000049511.71586.e0.

[25] TPC-H - Homepage. — Access mode: http://www.tpc.org/tpch.

[26] University Michael Stonebraker, Stonebraker Michael. The Case for Shared Nothing // Database Engineering. — 1986. — Vol. 9. — P. 4–9.

[27] Wikipedia. Session (computer science) — Wikipedia, The Free Encyclopedia. — 2017. — [Online; accessed 19-May-2017]. Access mode: https://en.wikipedia.org/w/index.php?title=Session_(computer_science)&oldid=769084675.

[28] Григорьев Валентин. Реализация операторов в распределенной дисковой колоночной СУБД : Выпускная квалификационная работа бакалавра / Валентин Григорьев ; Санкт-Петербургский Государственный Университет. — 2017.

[29] Ключиков Евгений. Исследование стратегий обработки запросов в распределенной дисковой колоночной СУБД : Бакалаврская работа / Евгений Ключиков ; Санкт-Петербургский Государственный Университет. — 2017.